On Admission Control for Profit Maximization of Networked Service Providers

Akshat Verma
akshatverma@in.ibm.com

Sugata Ghosal
gsugata@in.ibm.com

IBM India Research Laboratory
Block 1, Indian Institute of Technology
Hauz Khas, New Delhi 110016, INDIA.

ABSTRACT

Variability and diverseness among incoming requests to a service hosted on a finite capacity resource necessitates sophisticated request admission control techniques for providing guaranteed quality of service (QoS). We propose in this paper a service time based online admission control methodology for maximizing profits of a service provider. The proposed methodology chooses a subset of incoming requests such that the revenue of the provider is maximized. Admission control decision in our proposed system is based upon an estimate of the service time of the request, QoS bounds, prediction of arrivals and service times of requests to come in the short-term future, and rewards associated with servicing a request within its QoS bounds. Effectiveness of the proposed admission control methodology is demonstrated using experiments with a content-based messaging middleware service.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability and serviceability

Keywords

Admission control, Shortest Remaining Job First (SRJF), Profit maximization, Service level agreement (SLA), Quality of Service (QoS), Service time estimation, Short-term Prediction, Web Service

General Terms

Performance, Algorithms, Management

1. INTRODUCTION

Online businesses are struggling to cope with the ever-increasing maintenance and ownership costs of their computing resources. Outsourcing models (e.g., Application Service Providers (ASP), Web Services) use economies of scale and shared computing to reduce these costs. Ownership costs are also reduced because of the fact that if the peak load of all its customers do not coincide in time, then the aggregated peak-to-average ratio of shared resource utilization is significantly reduced. Hence, providers could only provision resources for a fraction of the combined peaks and service the customers for most of the time. In such a finite server capacity scenario, service-level agreements (SLAs) specifying certain quality of service (QoS) guarantees (e.g., average latency $< \tau msec$) need to be in place between a service provider and her clients. We envision that in emerging SLA-driven ASP model, requests would come with associated rewards and penalties. Penalties are incurred when the request is either not serviced at all or serviced outside the QoS bounds. Rewards, on the other hand, are obtained for servicing a request within QoS bounds. Realistically, as the resources of the service provider are less than the peak requirement, it becomes necessary to employ appropriate admission control measures to reject right subset of incoming requests so that the remaining requests can be serviced within their QoS bound.

Current web services[1] suffer from unpredictability of response time that is caused by the first-come-first-serve (FCFS) scheduling model and the "bursty" workload behavior. Thus, under heavy load scenario if a request with small resource requirement arrives at a web service later than requests with large resource requirements, it suffers from a long delay before being serviced, causing the *de facto* "denial-of-service" effect. Anecdotal evidence suggests that service time requirements of CGI-based requests or dynamic objects are one or more than one order of magnitude greater than that for static objects [1] in a conventional web hosting scenario. The same is true for bandwidth required to deliver multimedia (text, image, audio, video) objects to a remote client from a multimedia content delivery service. Recent studies show that though the CoV (coefficient of variation) of requests is as low as 1 to 2 for each type of object, the CoV of the aggregated web object type can be of the order of 10 [2], i.e., even though the web objects of one type may be similar, the aggregation of many types of web content makes the aggregated web object highly diverse. Hence, service time requirements for various incoming requests is expected to be highly diverse. Thus, service time requirements of an incoming request must be taken into account while taking the Admission Control (AC) decision for the request.

We propose in this paper a short-term prediction based online admission control methodology to maximize the profit of the provider. To effectively handle the variability and diverseness of incoming service requests via the web, the service time of a request is utilized as the

[1]Note that in this paper web service and networked service are used synonymously. Strictly speaking, a web service interface describes a collection of operations that are network accessible through standardized XML messaging.

WWW2003, May 20–24, 2003, Budapest, Hungary.
ACM 1-58113-680-3/03/0005.

most important criterion for making AC decisions. We prove that shortest remaining job first (SRJF) based request admission policy minimizes the penalty in an offline setting. The corresponding online version of the policy is derived using short-term prediction of arrivals of future requests, their service time distribution, and capacity availability, after accounting for the capacity usage of already admitted requests. Our proposed method, in this manner, derives its effectiveness by combining the complimentary strengths of off-line and online approaches. No assumption is, however, made regarding the nature of distributions of arrival rates and service times of incoming requests. Finally we present a modification of SRJF that attempts to maximize the profit of the provider taking into account rewards and penalties associated with the requests. Experimental results are presented to demonstrate the effectiveness of the proposed AC techniques for "bursty" workload conditions.

1.1 Related Work

Congestion control and QoS support in network transmission has been reported extensively. However, network-level QoS support is not sufficient in providing user perceivable performance without QoS support in web hosting servers, which can become heavily loaded and hence a bottleneck.

Conventional admission control schemes use a tail-dropping strategy to admit incoming requests. Under steady workload condition with similar service time requirements, queues of appropriate length along with careful capacity planning can be designed based on queueing theory models so that requests are admitted unless the queue is full. Chen and Li [3] use an essentially similar idea for AC in multimedia servers with mixed workloads. This method employs queues of different lengths for different object types, and a look-up table for online resource allocation (reservation) to adapt to workload changes. Fundamentally, variability and diversity among incoming requests tend to make *queue size ineffective* for making admission control decisions [2]. In PACERS (Periodical Admission Control based on Estimation of Request Rate and Service Time) algorithm [2], service time estimation is used to estimate the future capacity commitment for the already admitted (queued) requests. A new request is rejected if spare resource capacity (taking into consideration the resource requirement of queued requests) is not sufficient to service it. Thus, requests are rejected in a FIFO manner not taking into account the revenues (or service time) associated with those. Since even the requests belonging to the same class may be diverse in terms of their service time, it is necessary to choose between requests of the same class as well for maximizing the profit of the provider.

The profit maximization of service providers has been addressed by several researchers in context of self-similar, moderate workload condition (analogously, servers with sufficient capacity). Most of these address allocation of resources for various request classes with QoS guarantees so that resources are optimally utilized, thereby maximizing the profit of the providers. Among these, Liu *et al.* [4] proposed a multi-class queueing network model for the resource allocation problem in the offline setting, and solved it using a fixed-point iteration technique with the assumption that the average resource requirement for all the classes combined is less than the total available resource. The allocation of servers as discrete resources in a server farm has also been studied by Jayram *et al.* [5], keeping in mind the practical constraint that these need a finite time to be reallocated. A short-term prediction is used to make online allocation decisions, thus combining the principles of offline algorithms in an online setting. Work of similar nature has been carried in online call admission and bandwidth allocation which have been framed as cost minimization problem instead of revenue or benefit maximization [6, 7, 8, 9]. Chang *et al.* [10] adopted Irani's algorithm [11]. and addressed bandwidth allocation problem for web hosting based on resource availability and assigned revenue. However, Irani's algorithm has a competitive ratio of $O(\log R)$ (where R is the reward obtained by an optimal policy). Moreover, this method too does not present any intelligent method to choose between requests of the same class.

The key idea, used in our admission control approach, is our algorithm is not blind, as most online algorithms are, but uses an expected distributions of arrivals, and service times for future requests. This allows us to effectively handle admission control problem for request arrivals of "bursty" nature with widely varying service time requirements (exponential or even heavy tailed). Another important distinction in our approach is the *use of service time* to choose between different requests for admission (interestingly, a study conducted by Joshi [12] about resource management on the downlink of CDMA Packet data networks shows that algorithms which exploit request sizes seem to outperform those that do not).

1.2 Organization

We formulate the setting and explain our system design next. In Section 3.1, we explain our approach to the problem by relaxing some constraints and presenting an offline algorithm that maximizes the number of requests serviced. We then present an online version of the algorithm that maximizes the expected number of requests serviced within their QoS bounds by using the estimated distributions. In Section 3.2 we enhance the method to maximize the difference between the sum of expected rewards and the penalties. We present performance results of our algorithms for a publish-subscribe messaging service [17] in Section 4 and conclude with our observations in Section 5.

2. PROPOSED AC SYSTEM MODEL

The AC System controls the usage of some finite capacity resource (CPU, bandwidth etc.) that is used by the hosted service. It admits a subset of requests such that a pre-defined objective function is maximized. The concept of capacity of a resource is obvious in case of bandwidth [10], e.g., 128kbps. However, even for a computing service with resource like CPU, capacity can be naturally defined. For example, in a multi-threaded web-service, the maximum number of threads spawned for servicing the requests is bounded for issues of scale and performance. Hence, the number of concurrent threads or analogously the number of concurrent requests denotes the capacity.

2.1 Problem Formulation

We now give a concrete formulation of the profit maximization problem that our AC System solves. Assume an input set of n requests, where each request i can be represented as R$\{arrivalTime(a_i)$, $serviceTime(s_i)$, $reward(R_i)$,

penalty(P_i), *responseTimeBound*(b_i), *capacity*(c_i), *service-Class* (Cl_i)}. Define C as the total capacity of the resource available and T_{tot} as the total time under consideration. The problem then is to find a schedule $(x_{i,t})$ of requests such that the overall revenues are maximized. Formally, we have

$$max \quad \sum_{i=1}^{n}[R_i \sum_{t=1}^{T_{tot}} x_{i,t} - P_i(1 - \sum_{t=1}^{T_{tot}} x_{i,t})] \tag{1}$$

$$s.t. \quad \sum_{i=1}^{n} c_i p_{i,t} \leq C \qquad \forall \text{ time } t;$$
$$\sum_{t=1}^{T_{tot}} x_{i,t} \leq 1 \qquad \forall \text{ request } i;$$

$$x_{i,t} = \begin{cases} 1 & \text{if request } i \text{ is scheduled at time } t \\ & \text{and } (t + s_i - a_i) < b_i \\ 0 & \text{otherwise} \end{cases}$$

$$p_{i,t} = \begin{cases} 1 & \forall t : (t \in (\tau, \tau + s_i - 1) \\ & \text{and } x_{i\tau} = 1) \\ 0 & \text{otherwise} \end{cases}$$

$\sum_{t=1}^{T_{tot}} x_{i,t} = 0$ implies that the request is rejected. We note that this problem can be modeled as bandwidth allocation problem which is known to be NP-Hard even in a generalized off-line setting [14]. Moreover, the problem needs to be solved in an online setting where the decision of rejecting a request is made without knowledge of the requests which are scheduled to arrive later.

2.2 Reward and Penalty Model

A general reward and penalty model gives us the flexibility to optimize a variety of objective functions. A provider-centric model would have rewards proportional to service time. Also, in the case of differentiated services provided by the service provider, the rewards associated with a request would have the SLA class as one of the input parameter. If the SLAs have dynamic rewards, then the number of requests serviced would also be a parameter. An example for dynamic SLAs could be the following. If the provider services 90% of the requests then the reward and penalty per request is R_{90}, P_{90}. However, if the total number of requests serviced is greater than 90%, the penalty $P_{>90}$ may be less than P_{90}. Hence, reward (and penalty) can be expressed as a function $f(s_i, C_i, State_i)$ where

C_i = the class of the request i
$State_i$ = the state of the requests of the customer i.

A user-centric service provider may have a reward model that tries to enhance user experience. For example, the reward model could try to minimize the average waiting time of a user's request. The algorithm should not be dependent on the pecularities of the model and the admission controller should easily adapt to a new reward model. The reward function can be formulated in terms of any set of parameters and hence we may solve the cost minimization or QoS improvement problems by solving the appropriate profit maximization problem.

In practice, one may find that reward is only related to the SLA class. Moreover, if reward is found to be dependent on service time, the dependency is linear i.e., $R_i \propto s_i$ or sub-linear, since a request with longer service time consumes more resource and thus should be charged more. The scenario where reward is super-linear in service time, i.e., $R_i \propto (s_i)^c; \quad c > 1$, would be rare, as it would contradict the *Law of Diminishing Marginal Utility* [18]. Hence, an AC algorithm should be able to work well with sub-linear and linear reward models. As one may observe later, our algorithms are designed to work well with sub-linear reward models. Also, we report experiments which show that they also work well with linear reward models.

Since, by servicing a request, we are on an average expected to fetch reward proportional to the average service time of all requests, the penalty should also be related to this metric, at least in an expected sense. Also, for a request which has not been accepted, the actual service time can not be measured and the only metric available is the average. Thus, a realistic penalty model is one where the penalty for rejecting a request is proportional to average service time.

2.3 System Issues and Our Design

The problem of admission control of a web-based service with the objective of profit maximization poses challenges on both the system design front as well as algorithmic front. We follow a system design methodology that allows any algorithm to be plugged into the system. Similarly, the algorithm we design would be applicable to all admission control settings that can be formulated as the general maximization problem described earlier.

We will now look at a typical web-service to understand the system requirements. A typical web-service has multiple service instances running on different servers. Access to these instances is via a gateway that also implements the admission control functionality. The links between the gateway and the service instances may also have significant communication cost. Moreover, each service instance maintains a waiting queue for requests.

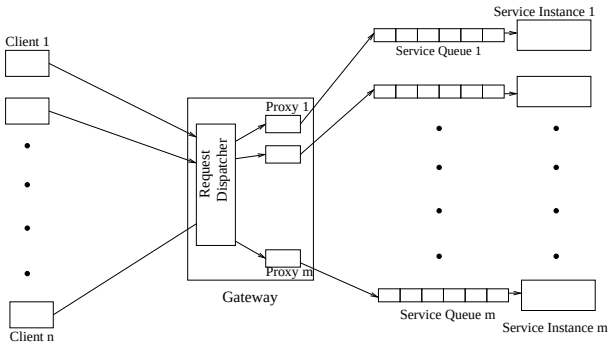

Figure 1: Admission Controller System Model

Hence, a possible AC strategy is to apply the tail-dropping strategy at each of the service instances. To elaborate further, the admission control policy would then be based on the queue length of the service instance I and implemented by the service instance instead of the gateway. This is a natural extension to the tail-dropping strategy that is employed in the existing web-servers. A more refined strategy could be based on Chen *et al.*'s work [2] by using the estimated system capacity needed by the queued requests. This important work presents a method to estimate the system capacity needed for the queued requests and shows that using system capacity instead of queue length leads to significant performance improvements.

However, delegating the admission control responsibility to the service instances has some significant drawbacks. Firstly, the different service instances may experience very different workloads and hence a local policy may not be globally optimal. Also, in case a request is

rejected, the client receives the information much later, i.e., after it has incurred the communication cost from the gateway to the service instance and back. This cost may be insignificant in a LAN but if the instances are geographically distributed this cost would be of concern. Moreover, if bandwidth is the constrained resource, then the request uses up bandwidth between the gateway and service instance.

We are now in a position to list the additional features our AC system should have over the current systems. In order to solve the precise problem described by Eqn. 1, (i) the AC needs spare capacity information of the entire duration from the service instance, (ii) service time, reward and penalty of the request and (iii) complete information about all the other requests. To solve problem (i), the gateway requests the spare capacity information from each service instance I whenever the *refresh criterion* is met. The *refresh criterion* is true if either the gateway has forwarded k requests to I or time T has passed since the last update, where k, T are modifiable parameters. We note that spare capacity estimation is done as described in [2]. In order to solve problem (ii), the gateway contains a proxy (See Figure 1) for every service instance that would unmarshall the request to determine the reward, penalty and request parameters. In order to estimate the service time, the proxy has a *Request-to-Service time Mapper(RSM)* that takes as input the request parameters and returns estimated service time. To tackle problem (iii), the proxy has a *predictor* that estimates the arrival rate and service time distribution for requests of each kind. This estimate can be used by the algorithm in place of actual request attributes. The proxy also has an *admission controller (AC)* that implements the admission control algorithm. The overall design of the proxy is described in Figure 2. Note that an *RSM* is also needed at each service instance for spare capacity estimation over a time horizon.

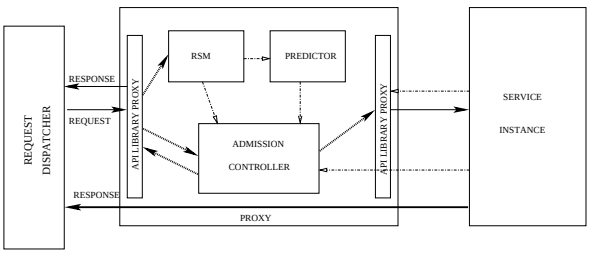

Figure 2: Layout of a Proxy

We now describe the control flow as a request r arrives. The request dispatcher forwards r to the appropriate proxy. The API proxy unmarshalls r and forwards the parameters to *RSM* and *AC*. *RSM* ascertains the service time from the request parameters and sends the service time associated with the request to *AC*. The *predictor* periodically sends the arrival rate and service time distribution of requests that are expected to arrive in the near future. The AC uses the above information and the spare capacity information to decide based on the methodology we describe later whether to accept the request. If the request is accepted, it is forwarded to the corresponding service instance. Otherwise, a "request rejected" response is sent to the client. Note that even though the decision to accept/reject a request is made by individual proxies, the dispatcher ensures that the workloads on different instances are similar.

2.3.1 Request to Service time Mapping (RSM)

We begin by distinguishing service time from response time. Service time of a request is the amount of time the resource (CPU or network bandwidth) is used by the request and not the end-to-end delay, which is the response time. Estimating the end-to-end response time for a web request to a reasonable accuracy is difficult, given the unreliable nature of the web. However, service time does not depend on the network delays and hence is unaffected by the nature of the web. We use the following method to estimate the service time of a multi-threaded web-service. For each thread which serves a request, a log is generated with the following information: 1) service parameters, 2) start time of the request and 3) end time of the request. A large number of diverse requests are sent to the web-service and enough logs are generated to make the data statistically reliable. Once the logs are generated, the service time distribution is computed for each value of the service parameters. The method is general enough to be used for the service time estimation for any other types of web-service implementations as well.

For any system, measured service time would then be a statistical function of request parameters and would be represented as $P_t(\tau|\hat{v})$ which is the conditional probability that $serviceTime = \tau$ given that request parameter vector $\hat{V} = \hat{v}$. The service for which our admission control system was built is a content-based publish/subscribe (pub/sub) messaging middleware [17]. Our experiments showed that the service time of the pub/sub service for any fixed number of attributes was Gaussian with the mean of the distribution linearly related to the number of attributes.

2.3.2 Prediction Methodology

The *predictor* makes a short-term forecast of arrival time distribution and service time distribution. It uses time-series analysis based short-term prediction for requests' arrival rate (λ_{tot}) as proposed in [1, 2, 13]. In practice, aggregated web traffic in short observation time windows is observed to be Poisson [1, 16], i.e.,

$$P(\text{inter-arrival delay} = x) = e^{-\lambda}\frac{\lambda^x}{x!} \quad (2)$$

The predictor also computes the distribution of the request attribute values ($P(\hat{v})$), based on past usage. It then uses these to compute the arrival rate ($\lambda_{\hat{v}}$) for each distinct parameter value vector($\hat{v}$) as

$$\lambda_{\hat{v}} = P(\hat{v}) * \lambda_{tot} \quad (3)$$

$P(\hat{v})$ for a publish-subscribe messaging utility [17] is observed to have a unique distribution. The number of attributes in a message is spread almost uniformly with small number of attributes (typically less than five) but decreases exponentially for larger numbers. The predictor uses the *parameter to service time mapping* $P_t(\tau|\hat{v})$ generated by the *request to service time mapper* and computes $P_s(t)$ (probability that a request would have service time t) according to

$$P_s(t) = \sum_{\forall \hat{v}} P(\hat{V} = \hat{v})P_t(t|\hat{v}) \quad (4)$$

The service time distribution $P_s(t)$ for the pub/sub messaging service [17] has a body, with a uniform distribution, and a tail, which follows a negative exponential distribution. The distribution was estimated using *Q.Q plot* and confirmed by the *Chi Square test.* However, it has been observed that the service time of web-traffic follows *heavy tailed distribution.* This property is surprisingly ubiquitous in the Web; it has been noted in the sizes of files requested by clients, the lengths of network connections, and files stored on servers [19, 20]. By heavy tails we mean that the tail of the empirical distribution function declines like a power law with exponent less than 2. i.e, if a random variable X follows a heavy-tailed distribution, then $P[X > x] \sim x^{-\alpha}, 0 < \alpha < 2$ where $f(x) \sim a(x)$ means that $\lim_{x\to\infty} f(x)/a(x) = c$, for some positive constant c. Hence, any practical admission control system should be able to perform well with a heavy-tailed service time distribution as well.

3. SRJF BASED AC ALGORITHMS

In this Section, we propose short-term prediction based AC algorithms for maximizing profit of the providers. We present a provably optimal offline algorithm for a special case of profit maximization problem in Section 3.1 and then extend it for the general case. The challenge in designing the algorithm is to come up with a practical solution. Since the number of requests n is very large, the algorithm for deciding whether a request r should be serviced, needs to be independent of n. Hence, we need a constant time algorithm for the NP-hard problem of deciding whether or not a request r should be serviced. Moreover, the problem has to be solved in an online setting.

3.1 Maximizing Number of Requests Serviced

We now simplify the maximization problem described by Eqn. 1 and find an offline solution to the simplified problem. We fix all the rewards and penalty as unity. Also, we assume that the $reponseTimeBound$ b_i is kept the same as $serviceTime$ s_i for each request i. Hence, Eqn. 1 reduces to

$$max \quad \sum_{i=1}^{n}((\sum_{t=1}^{T_{tot}} x_{i,t}) - (1 - \sum_{t=1}^{T_{tot}} x_{i,t})) \qquad (5)$$

with the same constraints. However, $x_{i,t} = 1$ can now hold only at $t = a_i$ because of the additional constraint on $responseTimeBound$. That is, we have the objective of maximizing the total number of requests serviced.

With complete information at hand, it is natural to service short jobs first(SJF). We, instead, use the shortest remaining time first that combines the idea of selecting a job, which is short and has fewer conflicting requests, and is used in operating system domain to minimize waiting time [15]. As we work with non-preemptive requests we refer to SRTF as Shortest Remaining Job First (SRJF) to avoid any confusion. The only difference of SRJF from SJF in this context is that the conflict set is restricted to the set of undecided requests, i.e., the requests which have neither been rejected nor serviced. The input to the algorithm is a list of requests, which we call undecided list and the output is a service list and a reject list. For example, assume the input request set is as shown in Figure 3(a), and the SRJF policy is followed. We take requests in an order, sorted by $arrivalTime$, i.e. r1, r2, r3, r4, r5, r6. We consider r1 first. Note that the only conflicting request with a shorter remaining time than r1 is r3. Also, even after servicing r3, we have spare capacity left for servicing r1. Hence, we accept r1. We then consider r2 and reject it because in the capacity left after serving r1, we cannot service both r2 and r3. By the shorter remaining time criterion r3 is preferred over r2. Hence, we reject r2 and serve r3. In the second set of requests again r4 is selected by a similar argument. But between r5 and r6, although r6 is shorter it ends after r5, and so r5 is selected. The output of the SRJF algorithm is given in Figure 3(b).

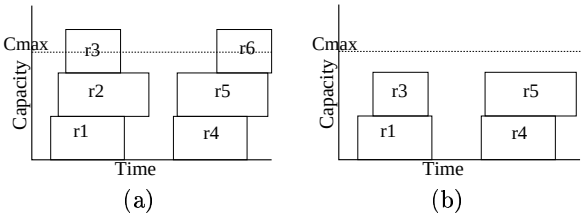

Figure 3: (a) Input Request Set and (b) Output Generated by SRJF

DEFINITION 1. *Shortest Remaining Job First Algorithm: We order the requests in order of their arrival. Then we service a request r_i if we have capacity left for r_i after reserving capacity for all the undecided requests r_j, s.t. $a_j + s_j \leq a_i + s_i$, Otherwise, we reject r_i.*

We note that once we have taken a request from the undecided list, we either accept it or reject it. It does not go back to the undecided list.

DEFINITION 2. *Define the conflict set of request i at time t (C_i^t) to be the set of all such requests r_j that have been not been rejected till time t and either (a) $a_i+s_i > a_j$ and $a_i+s_i < a_j+s_j$ or (b) $a_j+s_j > a_i$ and $a_i+s_i > a_j+s_j$.*

We now prove the central lemma of this Section.

LEMMA 1. *If SRJF rejects a request r_j in favor of r_i at time t, then $C_i^t \cup i \subseteq C_j^t \cup j$.*

PROOF. Note that since SRJF rejects r_j in favour of r_i, we have $a_i + s_i < a_j + s_j$. Let us assume there exists a request r_k s.t. $r_k \in (C_i^t - C_j^t)$. Also, since $a_i+s_i < a_j+s_j$, if case (a) is true for r_k to be in C_i^t, then $a_j+s_j < a_k+s_k$ and we have $a_i+s_i < a_k+s_k$. Similarly, if r_k is not in C_j^t by Case (a), i.e., we have $a_k > a_j + s_j$, then $a_k > a_i + s_i$. So, r_k is either in both C_j^t and C_i^t or in none. Now we look at Case (b). Since, r_k is in c_i^t, we have $a_i < a_k + s_k$. Also, since r_k is not in C_j^t, we have $a_k + s_k < a_j$, i.e., $a_i < a_k + s_k < a_j$. Also, since r_k is in the active set at time t, we also have $a_k + s_k > a_j$. Hence, we have a contradiction. Therefore no such r_k can exist. □

THEOREM 1. *If the capacity of the resource is unity, SRJF maximizes the total number of serviced requests.*

PROOF. Let us assume the optimal services a set $\mathbb{O}$ of requests and SRJF services a set $\mathbb{S}$. Then $\forall r_j \in \mathbb{O} - \mathbb{S}$ $\exists r_{j'} \in \mathbb{S}$ s.t. SRJF had accepted $r_{j'}$ in favour of r_j. By Lemma 1, we have $C_{j'}^t \cup j' \subseteq C_j^t \cup j$. Also, since r_j is rejected in favour of $r_{j'}$, we have $r_{j'} \in C_j^t$. Hence, we have $r_{j'} \in \mathbb{S} - \mathbb{O}$, as it conflicts with r_j and so $r_j \notin \mathbb{O}$. We now show that for any other $r_k \in \mathbb{O} - \mathbb{S}$, the corresponding

$r_{k'} \neq r_{j'}$. This follows from the fact that $r_k \notin C_j^t \cup j$ and $C_{j'}^t \cup j' \subseteq C_j^t \cup j$ and hence, $r_k \notin C_{j'}^t \cup j'$. Note that $r_a \in C_b^t \implies r_b \in C_a^t$ and hence $\forall r_i \in C_{j'}^t \cup j' : r_i \notin C_k^t \cup k$. Also, since $C_{k'}^t \cup k' \subseteq C_k^t \cup k$, we have $\forall r_i \in C_{j'}^t \cup j' : r_i \notin C_{k'}^t \cup k'$, i.e., $r_{j'} \neq r_{k'}$. Hence, $\forall r_j \in \mathbb{O} - \mathbb{S}$, $\exists$ distinct $r_i \in \mathbb{S} - \mathbb{O}$ hence, we have $|\mathbb{O} - \mathbb{S}| \leq |\mathbb{S} - \mathbb{O}|$, i.e., $|\mathbb{S}| \geq |\mathbb{O}|$, and as $|\mathbb{O}| \geq |\mathbb{S}|$, we have $|\mathbb{O}| = |\mathbb{S}|$ □

THEOREM 2. *SRJF maximizes the number of serviced requests for any capacity n.*

PROOF. (sketch): Theorem 1 can be naturally extended from capacity 1 to n. It can be shown on similar lines that for every request $r_j \in \mathbb{O} - \mathbb{S}$ there exists n requests in $\mathbb{S}$. Now all these $(n+1)$ requests cannot be serviced and optimal rejects at least one of the requests, i.e., for every request $r_j \in \mathbb{O} - \mathbb{S}$ there exists a request $r_{j'} \in \mathbb{S} - \mathbb{O}$. An argument similar to Theorem 1 using Lemma 1 shows that $r_{j'} = r_{k'}$ only if $r_j = r_k$. □

3.1.1 Online SRJF

The offline algorithm described above needs *a priori* information about a request's arrival and service time. Such information is, however, not available in a real admission control scenario. Also, the requests have a QoS bound on the response time and can be delayed only till the QoS bound is violated.

Hence, short-term prediction of requests' arrival rate and service time distribution is utilized to solve the request maximization problem in a practical online setting. Note that we can compute the service time of a request as well as the service time distribution using Eqn. 4. Also, the request arrival rate to a web service can be modeled as a distribution [1, 13](e.g., Poisson for web-traffic). No assumption is made about the type of distribution or its persistence in this paper.

Since the Shortest Remaining Job First (SRJF) algorithm takes requests sorted on their arrival times, it is easily transformed as an online algorithm. Our online SRJF algorithm then works in the following way. When a request arrives, it is checked whether this request can be serviced, given that the expected number of future requests which are expected to end before it are serviced. To illustrate further, if a request arrives at time t and has a service time of ten, we find the expected number of requests which will arrive at either of $(t+1)$, $(t+2,)$ .., $(t+9)$ and end before $(t+10)$, i.e., all those requests which are expected to be serviced before the current request ends. This ensures that we maximize the total number of requests serviced in an expected sense, i.e., if the assumed distribution is an exact set of requests, we should be maximizing the number of requests serviced. Moreover, if the request cannot be serviced immediately, the condition is rechecked after a rejection *till such a time that the response time bound may be violated.* To illustrate further, if a request with $a_i = T$, $s_i = 10$ and $b_i = 20$ can not be serviced immediately by the SRJF criteria, it gets reevaluated till time $(T+10)$, after which it is finally rejected. The pseudo-code of the algorithm is presented below.

We define
L = the mean capacity of the requests,
$Pr(i)$ = probability of event i happening.
E = random request with all parameters associated with the respective random variables
ρ = discount ratio

```
1   function schedule
2     for every element j in the available array A[1...d]
3       futureRequests[j] = L * Pr(s_E ≤ (d − j))
4       backlog = 0
5       for k = 1 to j
6         backlog = backlog+
            futureRequests[k] * Pr(s_E ≥ (j − k))
7       end-for
8       capacityLeft = available[j]−
          ρ * (backlog + futureRequests[j])
9       if(capacityLeft ≤ 1)
10        return false
11      end-if
12    end-for
13    return true
14  end function
```

The discount ratio ρ served two purposes. It captures the confidence in the prediction as well as a future discounting ratio. A predictor with a high error probability would have a ρ much less than 1 as the estimation of $futureRequests$ may be off margin. On the other hand, a sophisticated predictor would have $\rho \to 1$. For actual service deployment, the service provider should start with a default value of ρ depending on the predictor used and converge to the value optimal for her. Note also that in case a request r is rejected by *schedule* once, it is recalled till such a time when the QoS bound on the request r cannot be satisfied, if it is delayed any further.

3.2 Balanced SRJF for Profit Maximization

Our algorithm for the general case of Profit Maximization (i.e., one in which all rewards and penalties are not equal) is based on the following lemma about optimal $\mathbb{O}$.

LEMMA 2. *If for any request r_i, $\exists C_i' \subseteq C_i^t$ s.t.*
(i) $\forall r_j \in C_i'$
(a) $C_j^t \cap C_i' = \phi$, *(b)* $a_j + s_j < a_i + s_i$, *(c)* $r_j \notin \mathbb{O}$
(ii) $\sum_{r_j \in C_i'} R_j + P_j > R_i + P_i$, *then* $r_i \notin \mathbb{O}$.

PROOF. (sketch) Let us assume that such an r_i exists and belongs to optimal $\mathbb{O}$. Let us define $\mathbb{C}_i' = \cup_{r_j \in C_i'} C_j^t$. By a simple analysis for each element of $\mathbb{C}_i'$ on the lines of the proof of lemma 1, one can show that $\mathbb{C}_i' \subseteq C_i^t$. Hence, we can replace r_i by C_i' in $\mathbb{O}$ and the overall profit would be increased. Hence, we achieve a contradiction. □

Hence, our offline algorithm (offline BSRJF) essentially rejects all such requests r_i for which C_i' exists. It finds out all the candidate C_i' for each r_i and pre-reserves capacity for them. If spare capacity is left after pre-reservation, then r_i is serviced. This essentially leads to the following theorem about offline BSRJF.

THEOREM 3. *If offline BSRJF returns a set BS of requests, then $\forall S$ s.t $|BS - S| \leq 1$, the total revenue generated by BS is more than that generated by S.*

PROOF. The proof follows directly from Lemma 2 and the fact that *BSRJF* constructs BS *s.t.* $\forall r_i \in BS$, $\nexists C_i'$ □

In the online version of BSRJF, we compute the expected sum of reward and penalties for such C_i' candidates, i.e., we compute the sum of the expected rewards and penalties of non-conflicting request sets r_S that arrive later. We

compare this sum with the reward of the current request under consideration. If the sum of the expected rewards and the penalties saved exceeds the reward of the current request, we reserve capacity for r_S. This ensures that r_i is serviced only if there is no expected C_i' that would be rejected later.

This is incorporated by changing Line 3 of the online SRJF algorithm by

3 $futureRequests[j] = L* \sum_{i=1}^{d-j}(Pr(s_E = i) * f(d,i,j))$;

where,

$$f(d,i,j) = 1, \text{ if } \exists k \in \mathbb{N} : R_m \leq \tilde{R}_{s_i} + Pr((s_E = \tilde{s}) \leq (d-j-i)/k)) * k * (\tilde{R}_{\tilde{s}} + \tilde{P}_{\tilde{s}})$$

$\tilde{R}_s$ = Expected (average) reward for a request with serviceTime s

$\tilde{P}_s$ = Expected (average) penalty for a request with serviceTime s

Note that now capacity is not reserved for all earlier ending requests but only those who belong to a such a set C_i'. The algorithm in the offline scenario no longer guarantees an optimal solution but a solution that in some sense is locally optimal. This is because there is no single request r_i that can be replaced from the solution by some C_i' and the solution would improve. However, there may exist a set of such r_i's that could be removed and the objective function (Eqn. 1) may increase.

We take an example to explain the difference of this algorithm from online SRJF. If a request r_i of length 10 which starts at time-unit 50 and has successfully passed up to 55^{th} time-unit by the algorithm and there is a conflict with a request r_j, which spans from $55-58$, we may choose r_i if *the probability of a request of length two (which can fit in* $58-60$*) times the penalty of one request is less than the difference in net reward of* r_i *and* r_j. More precisely, resource is not reserved for r_j in favor of r_i if for an expected request k:

$$Pr(s_k \leq 2)(\tilde{R}_k + \tilde{P}_k) \leq R_i + P_i - (R_j + P_j) \quad (6)$$

Herein, a request of larger length r_i is admitted which may disallow an expected shorter request r_j later but the probability that there would be another request r_k in the remaining time is very low, i.e., expected sum of rewards and penalties of $C_i' = r_j \cup r_k$ is less than $R_i + P_i$. One may note that within this formulation *online SRJF* can be thought of as representing capacity for all such candidate C_i' irrespective of the revenue generated by C_i'. We now explain BSRJF's behaviour when the requests are from multiple SLA classes.

Multiple SLA Classes: When a choice has to be made between requests from the same class, *BSRJF* chooses the one that has a shorter remaining time. However, when a choice is to be made between requests of different SLA classes, *the actual measure implicitly used is reward per unit capacity consumed*. Note that a request of low reward class would have $f(d,i,j) = 1$ for all the expected requests of higher classes, i.e., *it would be rejected in favor of any shorter request of the higher priority classes irrespective of the fact that there may not be any chance of scheduling an extra request.* On the other hand, assuming zero penalty, a high priority request can be rejected in favor of low priority requests only if it is longer than the reward ratio of the two SLA classes times the expected service time of the low priority requests, i.e., *in a zero penalty scenario, where a request of Gold class has a reward three times the reward of a Silver class request, we would reject the gold request only if we expect to service three disjoint silver requests in the same time.*

3.3 Computational overheads

We now assess the overhead we have to incur for having an elaborate admission controller. For any client server system, a server thread would accept a request and add it to a service queue. Also, even a simple tail-dropping admission controller would have to look at the queue to determine its length and decide if the incoming request has to be dropped. Hence, the cost of admission control is at least $O(1)$ for every request which has arrived at the AC. Note also that in case the request is accepted, we have to add the request to the queue, which would imply scanning the whole request which is $O(l)$, where l = length of the encoding of the request. Hence, the running time per request is $O(l)$.

In our AC system, we parse the headers of all requests to determine its parameters. Once the parsing is done, the AC algorithm works only with an array (*available array* of length equal to the short-term forecast T), which is representative of the actual queue. Also, T can be suitably replaced by any constant k that is equal to the number of time-instances of T (values taken by j in the pseudo-code) at which we want to check the AC condition, e.g., if $T = 100$ and we make $k = 10$ equal sized partitions, we collapse all the entries between $10\ i + j,\ 0 < j < 10$ into a single entry corresponding to the i^{th} partition. One may reduce the number of points (in time) that one would check the AC condition for, to account for the fact that the load at different points would follow temporal correlation. Hence, if the AC condition is satisfied at a time t in the horizon, with high probability it would also satisfy in the temporal neighbourhood of t. This would bring the overhead even lower and could be assumed to be constant. Thus effectively the running time is $O(l)$ per request, which is the same as a tail-dropping AC system.

4. EXPERIMENTAL RESULTS

We conducted a large number of experiments with a hosted content-based publish-subscribe messaging middleware service [17]. Our experiments compare the performance of our SRJF-based algorithms to the following predictable service time-based AC algorithm.

Greedy: The greedy heuristic computes the expected system load over a time horizon by estimating the service time of all the requests which have been accepted (i.e., which are in the service queue). It accepts a request R, if there is spare capacity available to service R. Note that this heuristic is superior to the tail-dropping strategy, used by *current web-servers*. This is because greedy directly uses the system load information, whereas, tail-dropping systems use queue-length as an approximation of the system load. Performance numbers obtained in our experiments clearly establish the effectiveness of SRJF-based policies for admission control over greedy and, by extension, over tail-dropping as well.

The publish/subscribe messaging service [17] under consideration supports a programmatic interface, based on the standard Java Messaging Service (JMS) API, and allows for attribute-based subscription of published messages. One key QoS parameter for a pub/sub messaging service [17] is the latency of a message, i.e., the time a

message takes to reach a subscriber from a publisher via the messaging broker assuming zero network delay. The latency for a message is observed to vary with the CPU utilization of the broker in a manner, described in Figure 4. Hence, to ensure an upper bound on the latency guarantee as per the SLA, the CPU load should be kept below the threshold value. This places a limitation on the number of requests which can be serviced, as allowing any more requests would load the CPU above the threshold value and violate the latency bound resulting in losses for the publish/subscribe service provider. Thus, we can view the CPU as a resource which has a capacity equal to the threshold value which limits the number of concurrent requests. We instrumented the messaging service to obtain a performance model, and to determine the CPU thresholds to satisfy the QoS guarantees and the corresponding limit on the number of concurrent requests (Figures 4, 5), which is the capacity of the service.

4.1 Experimental Model and Parameters

The experiments simulate a request driven server model that hosts a publish-subscribe messaging service using empirical workload from the distributions generated as described in Section *2.3.2*. The traffic arrival was modeled as a Poisson process. Reward was proportional to the service time of the request and penalty was proportional to the average service time (Section 2.2). We used two service time distributions for our experiments. Firstly, we used the service time distribution unique to the messaging service, i.e., with an exponential tail. We also conducted simulations with a heavy-tail replacing the exponential tail to study the performance in other service time scenarios. Since we had no heavy-tailed data for the publish-subscribe messaging service, we used traces from Internet Traffic Archive. The trace used consisted of all the requests made in a one week to the Clarknet WWW server [21]. The service time distribution of the above trace fitted a log-normal distribution, which is heavy-tailed. For our experiments, the service time distribution was scaled to have a mean of 1000 μs. Each experiment was repeated until statistical stability was achieved and the computed means are reported.

We conducted experiments in the *single* and *multiple SLA class* settings. The performance improvement of our algorithms with *multiple SLA classes* is more significant than with *single SLA class*. Moreover, the performance improvement can be arbitrarily increased by increasing the reward ratio of the classes. The reason for this lies in the fact that *Greedy* is class blind. However, the significant performance improvement over *Greedy* that we achieve in a *single SLA class* scenario establishes that our algorithms do not depend on the reward ratio to achieve a superior performance. We would also like to note that these results are also valid if the competing algorithm is a proportionate greedy algorithm, one where a proportionate share of the resource is alloted to each class and that share is consumed greedily by the requests of that class, as the admission control algorithm for each of the partition of the resource could be interpreted as a greedy algorithm in isolation.

In Section 4.2, we report the performance of our algorithms against the greedy with varying inter-arrival delay, i.e., the request arrival rate decreases along the $x-$axis. In the second set of experiments we keep the arrival rates and mean service time constant and increase the variation in service time, which we call *spread*. The mean of the service time was fixed at 1 ms.

We observed that greedy performs worse when the tail is heavy as compared to a exponential tail. This can be attributed to the fact that the long requests become more frequent. However, our algorithms seem to show no adverse effect because of the heavy tail (Figures 9, 10). This is not surprising as our algorithms prune out the long requests even if they are more frequent. For lack of space, we report the full results with the exponential tail only and note that the improvement with a heavy tail is more significant (Figures 9, 10).

4.2 Effect of Request Arrival Rate on Performance

The results show (Figures 6, 8) that when the resource is in plenty (i.e., at low arrival rates or large inter-arrival delay), greedy performs as good as SRJF and BSRJF algorithm. This is expected because the requests are rejected very rarely, as at most times, spare capacity is available. However, as the request arrival-rate increases, the intelligent algorithms consistently beat the greedy heuristic by a large margin. In fact, we observe that the total number of requests serviced are more than doubled at a high request rate (low inter-arrival delay). These indicate that, as the number of admission control decisions increase, the performance improvement of our algorithm over greedy increases, emphasizing their superiority over greedy at heavy load. We note that BSRJF services approximately the same number of requests as SRJF. However, we note that the reward obtained by SRJF is less than BSRJF (Figures 6, 7). Hence, BSRJF gets a significant increase in rewards with only a small increase in penalty as compared to SRJF, thus increasing the revenue considerably.

The rewards obtained by the three heuristics implicitly contain the information about the capacity utilization by the heuristics. This is because reward is proportional to the service time of the request in the model used for these experiments. Hence, the sum of all the rewards obtained is directly proportional to the total system capacity used by all the serviced requests. It is also easy to see that the rewards obtained by the greedy is a good approximation for the total capacity in high load scenarios as greedy will almost always fully utilize the resource. In a low load scenario as well, the rewards obtained by greedy equal the maximum resource utilization as almost all the requests are serviced. Thus, Figure 7 shows how well the two heuristics consume the resource compared to greedy.

4.3 Effect of Service Time Variance on Performance

The number of requests serviced by our heuristics increases significantly with increase in spread as is evident from Figures 9 and 10. Since the mean service time and arrival rate are the same, the number of conflicts for resource should be comparable for all the data points. Hence, the only attribute, which increases, is the variability in the service time of the requests. As our algorithms take the service time as the most important criterion (SRJF uses it as the only criterion), the increase in variability allows our algorithm to make more intelligent decisions. This is reflected in the improved performance as opposed to greedy, which services approximately the same number of requests at all values of spread. A similar trend is visible (Figure 11) in the revenue obtained by balanced SRJF when compared with greedy. The increase

in spread increases the revenue of the balanced SRJF whereas it clearly shows no effect on the performance of greedy, which is blind to service time variability.

5. CONCLUSION

We propose a short-term prediction based admission control methodology for profit maximization of service providers and demonstrate its effectiveness under heavy and diverse workload conditions. Our methodology is less susceptible to workload variations, observed in Internet environment, since we do not assume any specific arrival and service time distribution. Our algorithms perform considerably better than greedy if either there is sufficient resource constraint, i.e., there are many more requests than the capacity available or the service times have a larger spread. The reason for the first is that there are actually more decisions to be made if there are more conflicting requests and hence, the selected requested set could be vastly different from the set that greedy generates. The second factor is an indicator of the diversity of the requests and hence, our algorithms perform better when the workload is more diverse.

6. ACKNOWLEDGEMENTS

The authors would like to thank Neeran Karnik for discussions about the system design and Rahul Garg for valuable comments on earlier versions.

7. REFERENCES

[1] A.K. Iyengar, M.S. Squillante, and L. Zhang. Analysis and characterization of large-scale Web Server Access Patterns and Performance. In *Proc. Conf. on World Wide Web*, 1999.

[2] X. Chen, P. Mohapatra, and H. Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proc. Conf. on World Wide Web,* 2001.

[3] I.-R. Chen, and S.T. Li. A Cost-Based Admission Control Algorithm for Handling Mixed Workloads in Multimedia Server Systems. In *Proc. IEEE Conf. on Parallel and Distributed Systems*, Kyongju City, Korea, 2001.

[4] Z. Liu, M.S. Squillante, and J.L. Wolf. On Maximizing Service-Level Agreement Profits. In *Proc. ACM Conf. on Electronic Commerce*, Orlando, FL, 2001.

[5] T.S. Jayram, T. Kimbrel, R. Krauthgamer, B. Schieber, and M. Sviridenko. Online Server Allocation in a Server Farm via Benefit Task Systems. In *Proc. ACM Symp. on Theory of Computing*, pp. 540–549, 2001.

[6] J.A. Garay, I.S. Gopal, S. Kutten, Y. Mansour, and M. Yung. Efficient online call control algorithms. In *Proc. Israeli Conf. on Theory of Computing and Systems*, pp. 285-293, 1993.

[7] R.J. Lipton, and A. Tomkins. On-Line interval scheduling. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1994.

[8] A. Bar-Noy, R. Canetti, S. Kutten, Y. Mansour, and B. Schieber. Bandwidth allocation with preemption. In *SIAM J. on Computing*, vol. 28, pp. 1806–1828, 1999.

[9] A. Bar-Noy, Y. Mansour, and B. Schieber. Competitive dynamic bandwidth allocation. *Proc. ACM Symp. on Principles of Distributed Computing*, 1998.

[10] Y.-C. Chang, X. Guo, T. kimbrel, and A. King. Optimal allocation policies for web hosting. In *Proc. Informs Conf. on Applied Probability*, New York, NY, 2001.

[11] S. Irani. Page replacement with multi-size pages and applications to web-caching. In *Proc. ACM Symp. on Theory of Computing*, El Paso, Texas, 1997.

[12] N. Joshi, S.R. Kadaba, S. Patel, and G.S. Sundaram. Downlink scheduling in CDMA data networks. In *Proc. Conf. on Mobile Computing and Networking*, pp. 179–190, 2000.

[13] M.S. Squillante, D.D. Yao, and L. Zhang. Web traffic Modeling and web server performance analysis. In *Proc. IEEE Conf. on Decision and Control*, pp. 4432-4437, 1999.

[14] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. (S.) Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. In *Proc. ACM Symp. on Theory of Computing*, pp. 735–744, Portland, OR, 2000.

[15] Siberschatz and Galvin. Operating System Concepts. *Addison-Wesley*, Fifth Edn., pp. 130–133, 1999.

[16] R. Morris, and D. Lin. Variance of aggregated web traffic. In *Proc. IEEE INFOCOM*, 2000.

[17] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra, Matching events in a content-based subscription system. In *Proc. Principles of Distributed Computing*, 1999.

[18] P.A. Samuelson and W.D. Nordhaus. Economics. *Tata McGraw-Hill*, pp. 80-83, 1998.

[19] M.F. Arlitt and C.L. Williamson. Internet web servers: Workload characterization and performance implications. In *IEEE/ACM Transactions on Networking*, 5(5):631-645, 1997.

[20] M.E. Crovella, M.S. Taqqu and Azer Bestavros. Heavy-tailed probability distributions in the World Wide Web. In *A Practical Guide To Heavy Tails*, pp. 3-26. Chapman & Hall, 1998.

[21] Clarknet HTTP Trace. Available at ftp://ita.ee.ibl.gov/traces/clarknet_access_log_Sep4.gz, Oct 15, 2002.

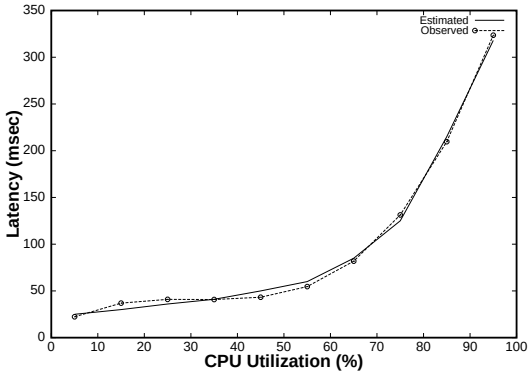

Figure 4: Median latency (Observed and Estimated by the model) vs. CPU Utilization for a messaging service.

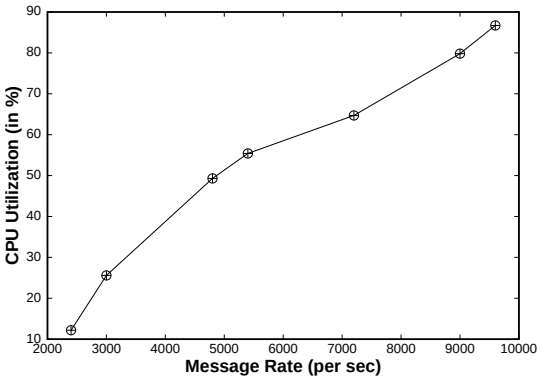

Figure 5: CPU Utilization vs. Request Rate for a messaging service.

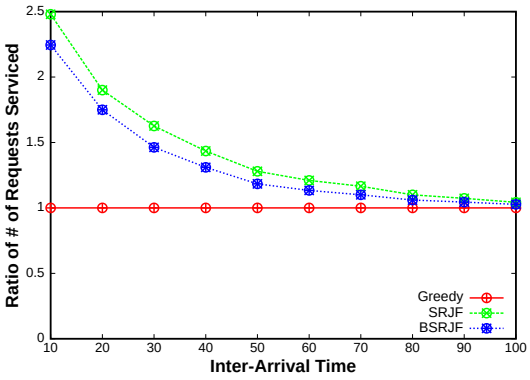

Figure 6: Requests Serviced by SRJF, BSRJF and Greedy with varying arrival rate

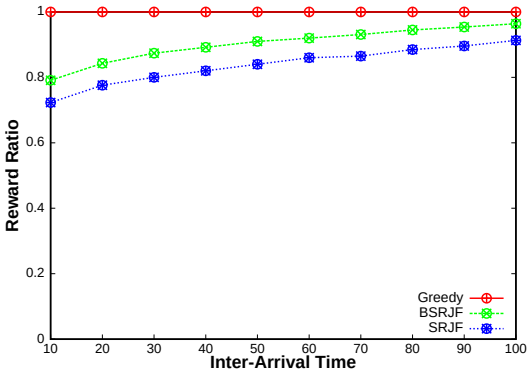

Figure 7: Rewards obtained by SRJF, BSRJF and Greedy with varying arrival rate

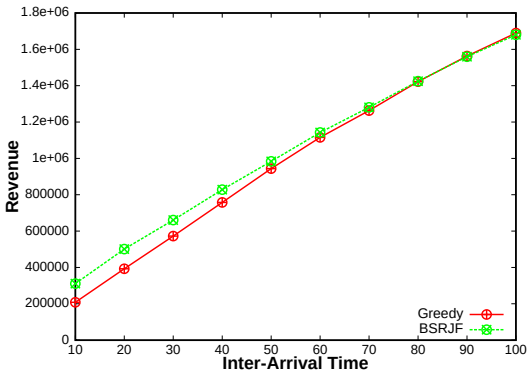

Figure 8: Revenue obtained by BSRJF and Greedy with varying arrival rate

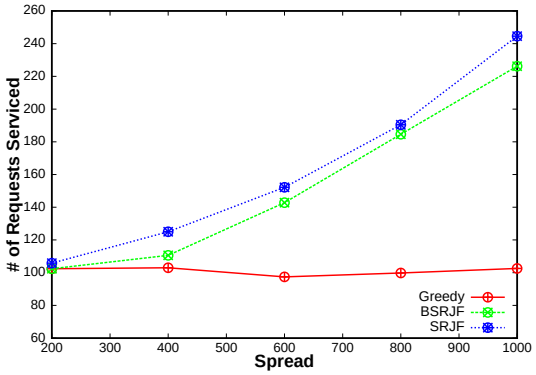

Figure 9: Requests serviced by BSRJF, SRJF and Greedy with varying service time spread for exponential service time

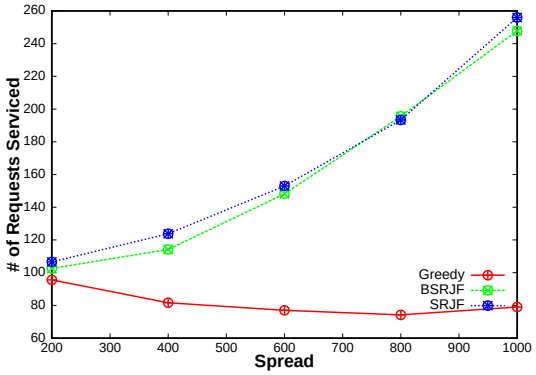

Figure 10: Requests serviced by BSRJF, SRJF and Greedy with varying service time spread for heavy-tailed service time

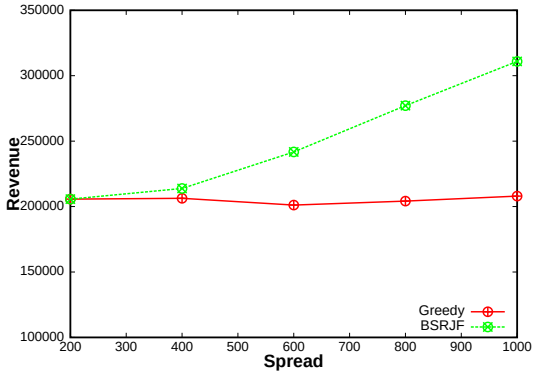

Figure 11: Revenue obtained by BSRJF and Greedy with varying service time spread

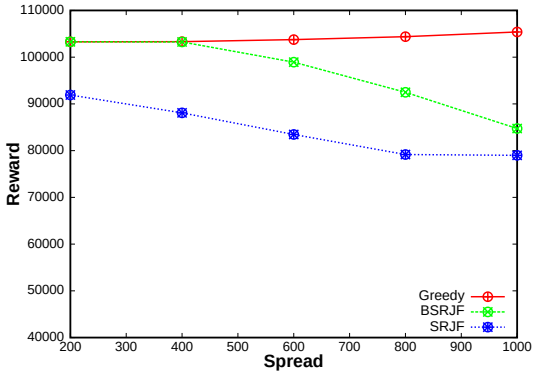

Figure 12: Reward obtained by BSRJF and Greedy with varying service time spread